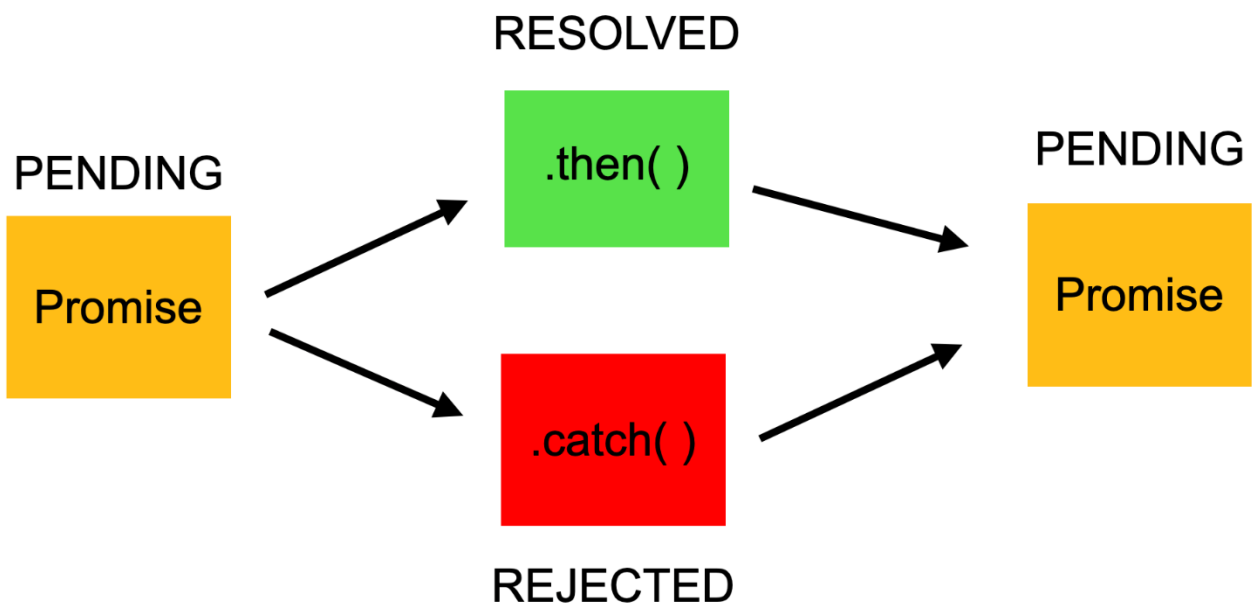
Promesas en JavaScript

Una Promesa es un objeto. Hay 3 estados del objeto Promesa:

- Pendiente: estado inicial, antes de que la promesa tenga éxito o falle
- Resuelto: promesa completada
- Rechazado: promesa fallida

Veamos una representación de los procesos de Promesas:



Cuando solicitamos datos del servidor mediante una Promesa, estará en modo pendiente hasta que recibamos nuestros datos.

Si logramos obtener la información del servidor, la Promesa se resolverá con éxito. Pero si no obtenemos la información, entonces la Promesa estará en el estado rechazado.

Además, si hay múltiples solicitudes, luego de que se resuelva (o rechace la primera Promesa), comenzará un nuevo proceso al que podemos adjuntarlo directamente mediante un método llamado **encadenamiento**.

La principal **diferencia** entre las funciones de **retrollamadas** y las **promesas** es que adjuntamos una retrollamada a una promesa en lugar de pasarla. Así que todavía usamos funciones de retrollamada (Callback) con Promesas, pero de una manera diferente (encadenamiento).

Encadenamiento:

Las funciones Callback se han utilizado solas para operaciones asincrónicas en JavaScript durante muchos años. Pero en algunos casos, usar Promesas puede ser una mejor opción.

Si hay varias operaciones asincrónicas por hacer y si tratamos de usar retrollamadas antiguas para ellas, nos encontraremos rápidamente dentro de una situación llamada "Infierno Retrollamada":

```
firstRequest(function(response) {
    secondRequest(response, function(nextResponse) {
        thirdRequest(nextResponse, function(finalResponse) {
            console.log('Final response: ' + finalResponse);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Sin embargo, si manejamos la misma operación con Promesas, ya que podemos adjuntar retrollamadas en lugar de pasarlas, esta vez el mismo código anterior parece mucho más limpio y fácil de leer:

```
firstRequest()
  .then(function(response) {
    return secondRequest(response);
}).then(function(nextResponse) {
    return thirdRequest(nextResponse);
}).then(function(finalResponse) {
    console.log('Final response: ' + finalResponse);
}).catch(failureCallback);
```

El código anterior muestra cómo se pueden encadenar múltiples retrollamadas una tras otra. El encadenamiento es una de las mejores características de Promesas.

Creación y uso de una promesa paso a paso

En primer lugar, usamos un constructor para crear un objeto Promesa:

```
const myPromise = new Promise();
```

Se necesitan dos parámetros, uno para el éxito (resolver) y otro para el error (rechazar):

```
const myPromise = new Promise((resolve, reject) => {
    // condition
```

```
});
```

Finalmente, habrá una condición. Si se cumple la condición, la Promesa se resolverá, de lo contrario será rechazada:

```
const myPromise = new Promise((resolve, reject) => {
    let condition;

    if(condition is met) {
        resolve('Promise is resolved successfully.');
    } else {
        reject('Promise is rejected');
    }
});
```

Ya está creada una promesa, vamos a usarla:

then( ) para promesas resueltas:

En la imagen del principio se observa que hay 2 casos: uno para promesas resueltas y otro para rechazadas. Si la Promesa se resuelve (caso de éxito), algo sucederá después (depende de lo que hagamos con la Promesa exitosa).

```
myPromise.then();
```

Se llama al método then( ) después de que se resuelva la Promesa . Entonces podemos decidir qué hacer con la Promesa resuelta.

Por ejemplo, registremos el mensaje en la consola lo que obtuvimos de la Promesa:

```
myPromise.then((message) => {
    console.log(message);
});
```

catch( ) para Promesas rechazadas:

Sin embargo, el método then( ) es solo para Promesas resueltas. ¿Qué pasa si la Promesa falla? Entonces, necesitamos usar el método catch( ) .

Del mismo modo que adjuntamos el método then( ). También podemos adjuntar directamente el método catch() justo después de then( ):

```
myPromise.then((message) => {
```

```
    console.log(message);
}).catch((message) => {
    console.log(message);
});
```

Entonces si la promesa es rechazada, saltará al método catch() y esta vez veremos un mensaje diferente en la consola.

Traducido y adaptado del artículo de Cem Eygi - JavaScript Promise Tutorial: Resolve, Reject, and Chaining in JS and ES6.